Answers

Solutions and Hints for Selected Exercises

Chapter 1

Section 1.1

5. To prove that two sets are equal, we must show that an element is in the first set if and only if it is in the second. Suppose $x \in \overline{S_1 \cup S_2}$. Then $x \notin S_1 \cup S_2$, which means that x cannot be in S_1 or in S_2, that is $x \in \overline{S_1} \cap \overline{S_2}$. Conversely, if $x \in \overline{S_1} \cap \overline{S_2}$, then x is not in S_1 and x is not in S_2, that is $x \in \overline{S_1 \cup S_2}$.

6. This can be proven by an induction on the number of sets. Let $Z = S_1 \cup S_2 ... \cup S_n$. Then $S_1 \cup S_2 ... \cup S_n \cup S_{n+1} = Z \cup S_{n+1}$. By the standard DeMorgan's law

$$\overline{Z \cup S_{n+1}} = \overline{Z} \cap \overline{S}_{n+1}.$$

With the inductive assumption, the relation is true for up to n sets, that is,

$$\overline{Z} = \overline{S}_1 \cap \overline{S}_2 \cap ... \overline{\cap S}_n.$$

Therefore

$$\overline{Z \cup S_{n+1}} = \overline{S}_1 \cap \overline{S}_2 \cap ... \overline{\cap S}_n \cap \overline{S}_{n+1},$$

completing the inductive step.

8. Suppose $S_1 = S_2$. Then $S_1 \cap \overline{S}_2 = \overline{S_1} \cap S_2 = S_1 \cap \overline{S}_1 = \varnothing$ and the entire expression is the empty set. Suppose now that $S_1 \neq S_2$ and that there is an element x in S_1 but not in S_2. Then $x \in \overline{S}_2$ so that $S_1 \cap \overline{S}_2 \neq \varnothing$. The complete expression can then not be equal to the empty set.

12. If x is in S_1 and x is in S_2, then x is not in $(S_1 \cup S_2) - S_2$. Because of this, a necessary and sufficient condition is that the two sets be disjoint.

15. (c) Since

$$\frac{n!}{n^n} = \frac{n}{n}\frac{n-1}{n}\cdots\frac{2}{n}\frac{1}{n}$$

is the product of factors less than or equal one. Therefore, $n! = O(n^n)$.

27. An argument by contradiction works. Suppose that $2 - \sqrt{2}$ were rational. Then

$$2 - \sqrt{2} = \frac{n}{m}$$

gives

$$\sqrt{2} = \frac{2m - n}{m}$$

contradicting the fact that $\sqrt{2}$ is not rational.

29. By induction. Suppose that every integer less than n can be written as a product of primes. If n is a prime, there is nothing to prove, if not, it can be written as the product

$$n = n_1 n_2$$

where both factors are less than n. By the inductive assumption, they both can be written as the product of primes, and so can n.

Section 1.2

2. Many string identities can be proven by induction. Suppose that $(uv)^R = v^R u^R$ for all $u \in \Sigma^*$ and all v of length n. Take now a string of length $n + 1$, say $w = va$. Then

$$\begin{aligned}(uw)^R &= (uva)^R \\ &= a\,(uv)^R, \text{ by the definition of the reverse} \\ &= av^R u^R, \text{ by the inductive assumption} \\ &= w^R u^R.\end{aligned}$$

By induction then, the result holds for all strings.

4. Since *abaabaaabaa* can be decomposed into strings *ab, aa, baa, ab, aa*, each of which is in L, the string is in L^*. Similarly, *baaaaabaa* is in L^*. However, there is no possible decomposition for *baaaaabaaaab*, so this string is not in L^*.

10. (d) We first generate three a's, then add an arbitrary number of a's and b's anywhere.

$$S \to AaAaAaA$$
$$A \to aA\,|bA|\,\lambda$$

The first production generates three a's. The second can generate any number of a's and b's in any position. This shows that the grammar can generate any string $w \in \{a,b\}^*$ as long as $n_a(w) \geq 3$.

11.

$$S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow ababS$$

from which we see that

$$L(G) = \{(ab)^n : n \geq 0\}.$$

13. (a) Generate one b, then an equal number of a's and b's, finally as many more b's as needed.

$$S \to AbA$$
$$A \to aAb|\lambda$$
$$B \to bB|\lambda$$

13. (d) The answer is easier to see if you notice that

$$L_4 = \left\{a^{m+3}b^m : m \geq 0\right\}.$$

This leads to the easy solution

$$S \to aaaA$$
$$A \to aAb|\lambda$$

14. (b) The problem is simplified if you break it into two cases, $|w| \bmod 3 = 1$ and $|w| \bmod 3 = 2$. The first is covered by

$$S_1 \to aaaS_1|a,$$

the second by

$$S_2 \to aaaS_2|aa.$$

The two can be combined into a single grammar by

$$S \to S_1|S_2.$$

16. (a) We can use the trick and results of Example 1.13. Let L_1 be the language in Example 1.13 and modify that grammar so that the start symbol is S_1. Consider then a string $w \in L$. If this string start with an a, then it has the form $w = aw_1$, where $w_1 \in L_1$. This situation can be taken care of by $S \to aS_1$. If it starts with a b, it can be derived by $S \to S_1 S$.

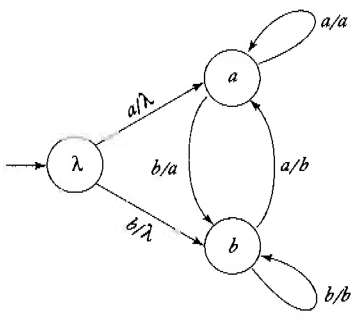
Section 1.3

1.

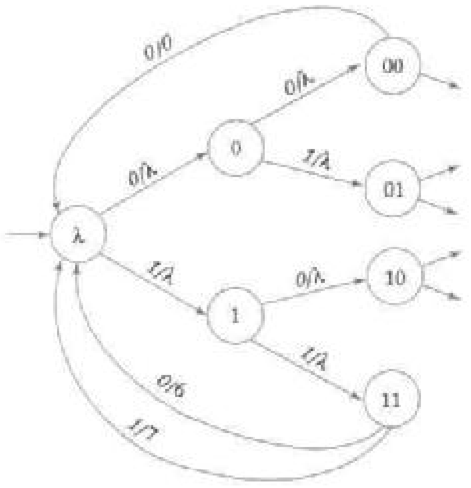
$$\begin{aligned} \text{integer} &\to \text{sign magnitude} \\ \text{sign} &\to + \mid - \mid \lambda \\ \text{magnitude} &\to \text{digit} \mid \text{digit magnitude} \\ \text{digit} &\to 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

This can be considered an ideal version of C, as it puts no limit on the length of an integer. Most real compilers, though, place a limit on the number of digits.

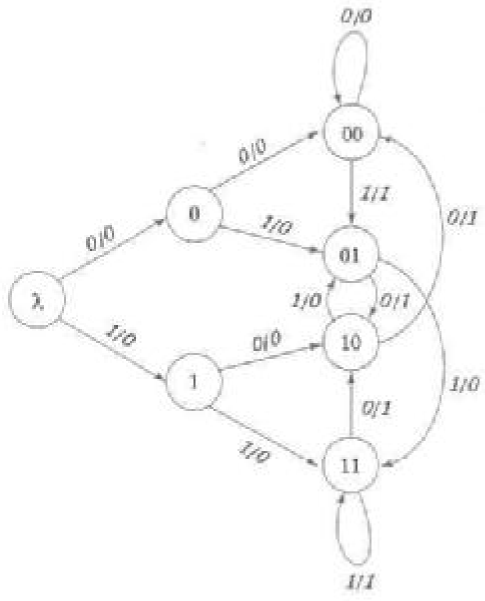
7. The automaton has to remember the input for one time period so that it can be reproduced for output later. Remembering can be done by labeling the state with the appropriate information. The label of the state is then produced as output later.



10. We remember input by labeling the states mnemonically. When a set of three bits is done, we produce output and return to the beginning to process the next three bits. The following solution is partial, but the completion should be obvious.



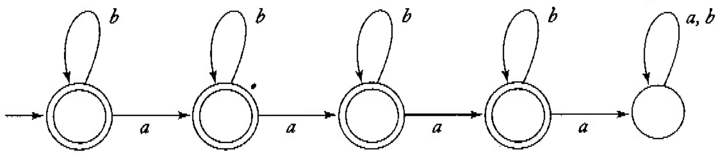
11. In this case, the transducer must remember the two preceding input symbols and make transitions so that the needed information is kept track of.



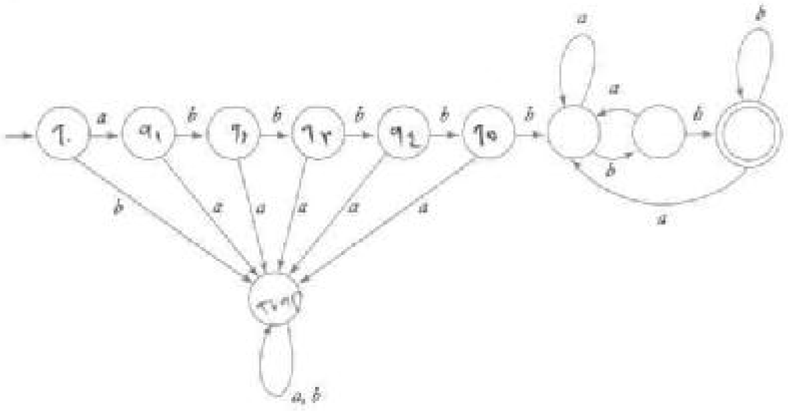
Chapter 2

Section 2.1

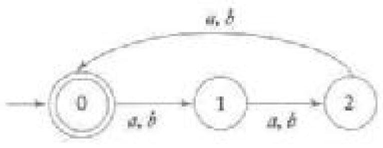
2. (c) Break it into three cases each with an accepting state: no a's, one a, two a's, three a's. A fourth a will then send the dfa into a non-accepting trap state. A solution:



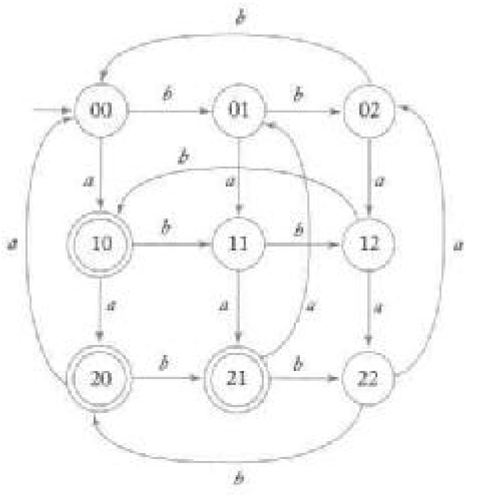
5. (a) The first six symbols are checked. If they are not correct, the string is rejected. If the prefix is correct, we keep track of the last two symbols read, putting the dfa in an accepting state if the suffix is bb.



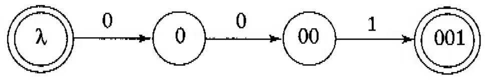
7. (a) Use states labeled with $|w| \bmod 3$. The solution then is quite simple.



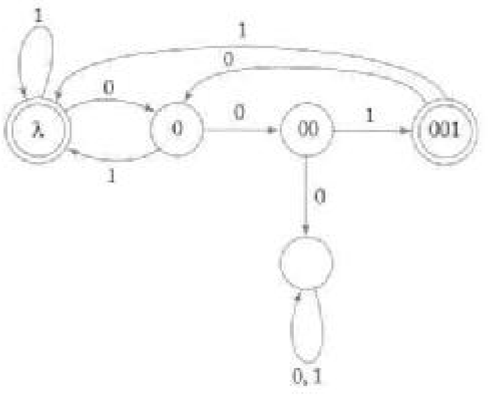
(d) For this we use nine state, with the first part of each label $n_a(w) \bmod 3$, the second part $n_b(w) \bmod 3$. The transitions and the final states are then simple to figure out.



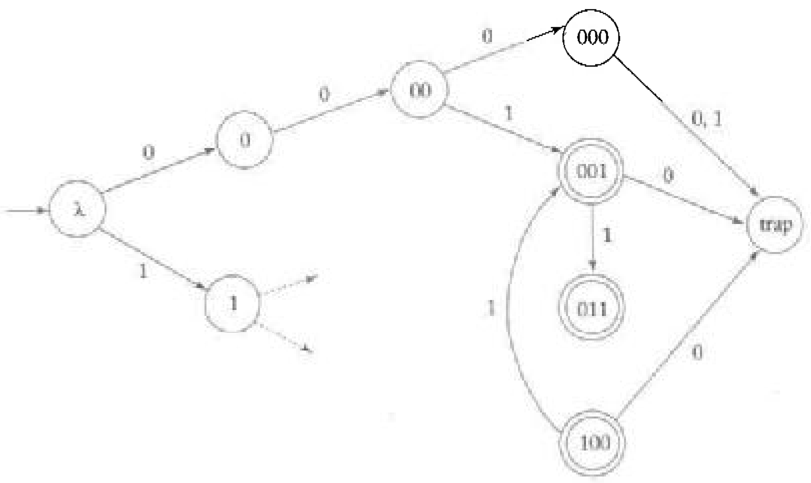
9. (a) Count consecutive zeros, to get the main part of the dfa.



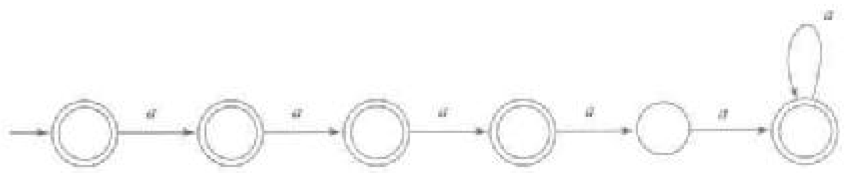
Then put in additional transitions to keep track of consecutive zeros and to trap unacceptable strings.



(d) Here we need to remember all combinations of three bits. This requires 8 states plus some start-up. The solution is a little long but not hard. A partial sketch of the solution is below.



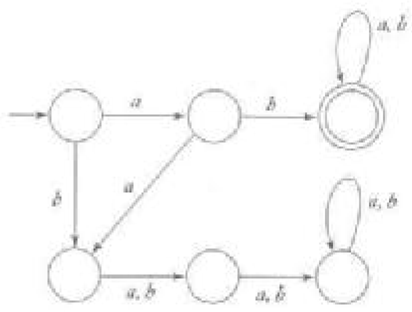
13. The easiest way to solve this problem is to construct a dfa for $L = \{a^n : n = 4\}$, then complement the solution.



21. (a) By contradiction. Suppose G_M has no cycles in any path from the initial state to any final state. Then every walk has a finite number of steps, and so every accepted string has to be of finite length. But this implies that the language is finite.

(b) Also by contradiction. Assume that G_M has some cycle in a path from the initial state to some accepting state. We can then use the cycle to generate an arbitrarily long walk labeled with an accepted string. But a finite language cannot contain arbitrarily long strings.

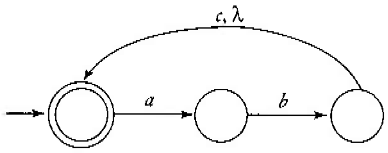
24. There are many different solutions. Here is one of them.



Section 2.2

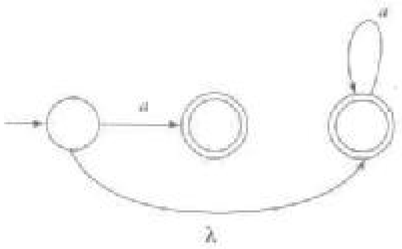
4. $\delta^* (q_0, a) = \{q_0, q_1, q_2\}, \delta^* (q_1, \lambda) = \{q_0, q_2\}$.

7. A four-state solution is trivial, but it takes a little experimenting to get a three-state one. Here is one answer:



8. No. The string abc has three different symbols and there is no way this can be accepted with fewer than three states.

15. This is the kind of problem in which you just have to try different ways. Probably most of your tries will not work. Here is one that does.

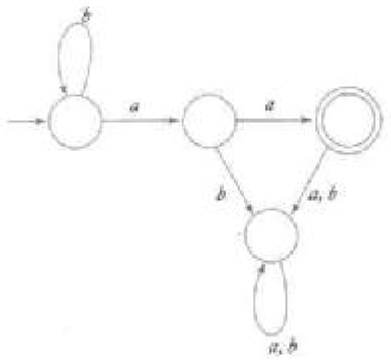


17. Introduce a single starting state p_0. Then add a transition

$$\delta (p_0, \lambda) = Q_0.$$

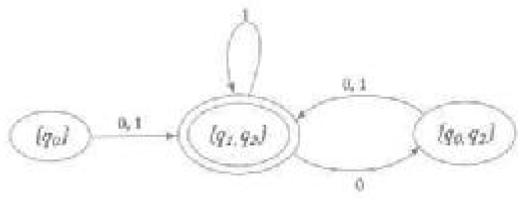
Next, remove starting state status from Q_0. It is straightforward to see that the new nfa is equivalent to the original one.

20. Introduce a non-accepting trap state and make all undefined transitions to this new state. Solution:



Section 2.3

2. Just follow the procedure nfa_to_dfa. This gives the dfa



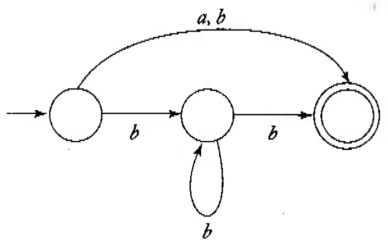
7. Introduce a new final state p_f and for every $q \in F$ add the transitions

$$\delta(q, \lambda) = \{p_f\}.$$

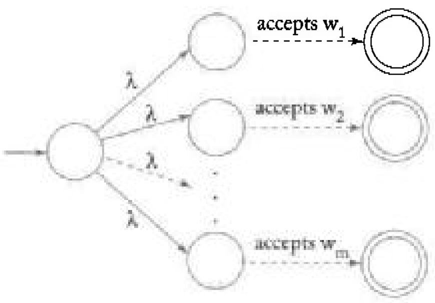
Then make p_f the only final state. It is a simple matter then to argue that if $\delta^*(q_0, w) \in F$ originally, then $\delta^*(q_0, w) = \{p_f\}$ after the modification, so both the original and the modifies nfa's are equivalent.

Since this construction requires λ-transitions, it cannot be made for dfa's. Generally, it is impossible to have only one final state in a dfa, as can be seen by constructing dfa's that accept $\{\lambda, a\}$.

8. Getting an answer requires some thought. One solution is

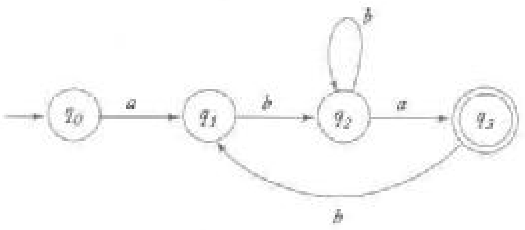


11. Suppose that $L = \{w_1, w_2, ...w_m\}$. Then the nfa

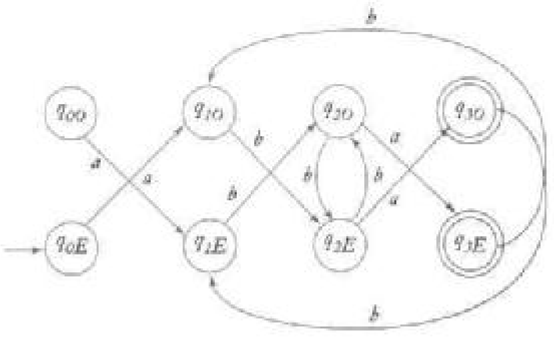


accepts L, so the language is regular.

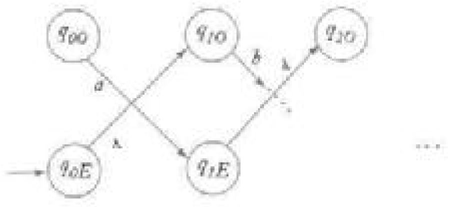
14. This is not easy to see. The trick is to use a dfa for L and modify it so that it remembers if it has read an even or an odd number of symbols. This can be done by doubling the number of states and adding O or E to the labels. For example, if part of the dfa is



its equivalent becomes



Now replace all transitions from an E state to an O state with λ-transitions.



With a few examples you should be able to convince yourself that if the original dfa accepts $a_1a_2a_3a_4$, the new automaton will accept $\lambda a_2\lambda a_4 ...$, and therefore *even* (L).

15. Suppose we have a dfa that accepts L. We then

(a) identify all states $\overline{Q}$ that can be reached from q_0, reading any two-symbol prefix v, that is

$$\overline{Q} = \{q \in Q : \delta^* (q_0, v) = q\}.$$

(b) introduce a new initial state p_0 and add

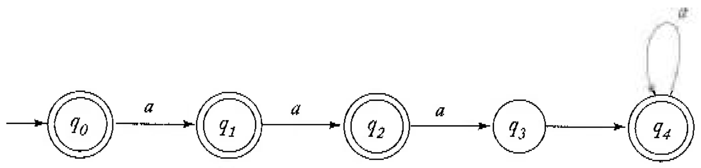
$$\delta (p_0, \lambda) = \overline{Q}.$$

It should not be hard to see that the new nfa accepts *chop*2 (L).

Although the construction is plausible, a complete answer requires a proof of the last statement.

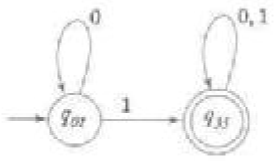
Section 2.4

2. (c)



This is minimal for the following reason. $q_3 \notin F$ and $q_4 \in F$, so q_3 and q_4 are distinguishable. Next, $\delta^*(q_2, a) \notin F$ and $\delta^*(q_4, a) \in F$, so q_2 and q_4 are distinguishable. Similarly, $\delta^*(q_1, aa) \notin F$ and $\delta^*(q_3, aa) \in F$, so q_1 and q_3 are distinguishable. Continuing this way, we see that all states are distinguishable and therefore the dfa is minimal.

4. First, remove the inaccessible states q_2 and q_4. Then use the procedure *mark* to find the indistinguishable pairs (q_0, q_1) and (q_3, q_5). This then gives the minimal dfa.



6. By contradiction. Assume that $\widehat{M}$ is not minimal. Then we can construct a smaller dfa $\widetilde{M}$ that accepts $\overline{L}$. In $\widetilde{M}$, complement the final state set to give a dfa for L. But this dfa is smaller than M, contradicting the assumption that M is minimal.

10. By contradiction. Assume that q_b and q_c are indistinguishable. Since q_a and q_b are indistinguishable and indistinguishability is an equivalence relation (Exercise 7), q_a and q_c must be indistinguishable.

Chapter 3

Section 3.1

2. Yes, because $((0+1)(0+1)^*)^*$ denotes any string of 0's and 1's. So does $(0+1)^*$.

5. (a) Separate into cases $m = 0, 1, 2, 3$. Generate 4 or more a's, followed by the requisite number of b's. Solution: $aaaaa^*(\lambda + b + bb + bbb)$.

(c) The complement of the language in 5(a) is harder to find. A string is not in L if it is of the form $a^n b^m$, with either $n < 4$ or $m > 3$, but this does not completely describe $\overline{L}$. We must also take in the strings in which a b is followed by an a. Solution:

$$(\lambda + a + aa + aaa)\, b^* + a^* bbbbb^* + (a+b)^*\, ba\, (a+b)^*.$$

9. Split into three cases: $m = 1$, $n \geq 3$, $n \geq 2$, $m \geq 2$, and $n = 1$, $m \geq 3$. Each case has a straightforward solution.

12. Enumerate all cases with $|v| = 2$ to get

$$aa\,(a+b)^*\, aa + ab\,(a+b)^*\, ab + ba\,(a+b)^*\, ba + bb\,(a+b)^*\, bb.$$

14. (c) You just have to get in each symbol at least once. The term

$$(a+b+c)^*\, a\,(a+b+c)^*\, b\,(a+b+c)^*\, c\,(a+b+c)^*$$

will do this, but is not enough since the a will precede the b, etc. For the complete solution you must generate all permutations of the three symbols, giving six terms that can be added. The answer, although quite long, is conceptually not hard.

15. (c) Create two 0's, interspersed with 1's, then repeat. But don't forget the case when there are no 0's at all. Solution: $(1^*01^*01^*)^* + 1^*$.

16. (a) Create all strings of length three and repeat. A short solution is $((a+b+c)\,(a+b+c)\,(a+b+c))^*$.

18. (c) The statement

$$(r_1 + r_2)^* \equiv (r_1{}^* r_2{}^*)^*$$

is true. By the given rules $(r_1 + r_2)^*$ denotes the language $(L(r_1) \cup L(r_2))^*$, that is the set of all strings of arbitrary concatenations of elements of $L(r_1)$ and $L(r_2)$. But $(r_1{}^* r_2{}^*)^*$ denotes $((L(r_1))^*\,(L(r_2))^*)^*$, which is the same set.

21. The expression for an infinite language must involve at least one starred subexpression, otherwise it can only denote finite strings. If there is one starred subexpression that denotes a non-empty string, then this string can be repeated as often as desired and therefore denote arbitrarily long strings.

23. A closed contour will be generated by an expression r if and only if $n_l(r) = n_r(r)$ and $n_u(r) = n_d(r)$.

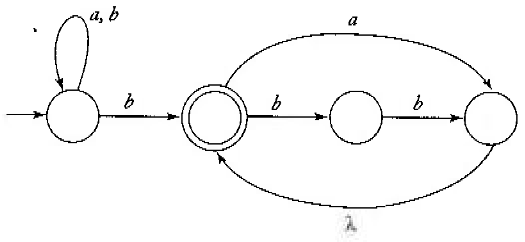
25. Notice several things. The bit string must be at least 6 bits long. If it is longer than 6 bits, its value is at least 64, so anything will do. If it is exactly 6 bits, then either the second bit from the left (16) or the third bit from the left (8) must be 1. If you see this, then the solution

$$(111 + 110 + 101)(0 + 1)(0 + 1)(0 + 1) +$$
$$1(0 + 1)(0 + 1)(0 + 1)(1 + 0)(1 + 0)(1 + 0)(1 + 0)^*$$

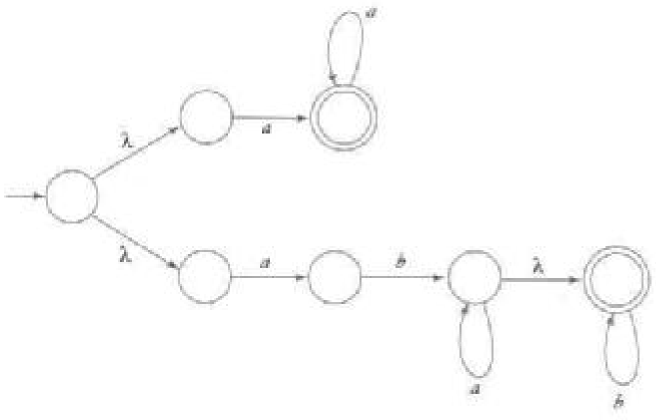
readily suggests itself.

Section 3.2

3. This can be solved from first principles, without going through the regular expression_to_nfa construction. The latter will of course work, but gives a more complicated answer. Solution:

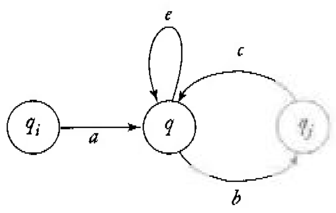


4. (a) Start with

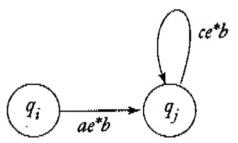


Then use the nfa_to_dfa algorithm in a routine manner.

7. One case is

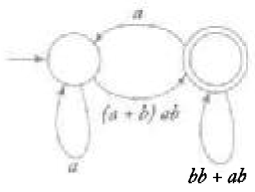


Since there is no path from q_j to q_i, the edges in the general case created by such a path are omitted. The result, gotten by looking at all possible paths, is



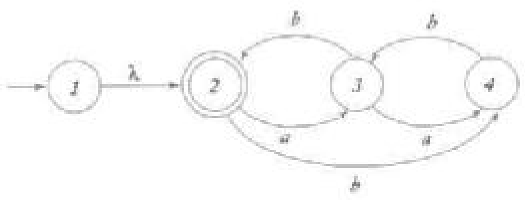
The other case can be analyzed in a similar manner.

8. Removing the middle vertex gives

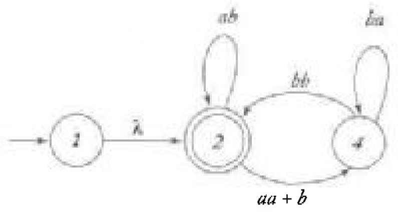


The language accepted then is $L(r)$ where $r = a^*(a+b)ab(bb+ab+aa^*(a+b)ab)^*$.

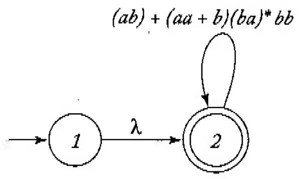
10. (b) First, we have to modify the nfa so that it satisfies the conditions imposed by the construction in Theorem 3.2, one of which is $q_0 \notin F$. This is easily done.



Then remove state 3.

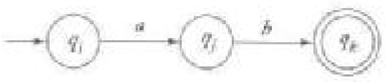


Next, remove state 4.

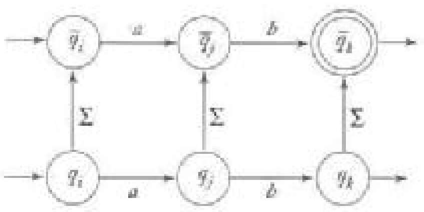


The regular expression then is $r = \left(ab + (aa+b)(ba)^* bb^*\right)^*$.

17. (a) This is a hard problem until you see the trick. Start with a dfa with states $q_0, q_1, ...,$ and introduce a "parallel" automaton with states $\overline{q}_0, \overline{q}_1, ...$ Then arrange matters so that the spurious symbol nondeterministically transfers from any state of the original automaton to the corresponding state in the parallel part. For example, if part of the original dfa looks like



then the dfa with its parallel will be an nfa whose corresponding part is



It is not hard to make the argument that the original dfa accepts L if and only if the constructed nfa accepts *insert* (L).

Section 3.3

4. Right linear grammar:

$$\begin{aligned} S &\to aaA \\ A &\to aA|B \\ B &\to bbbC \\ C &\to bC|\lambda \end{aligned}$$

Left linear grammar:

$$\begin{aligned} S &\to Abbb \\ A &\to Ab|B \\ B &\to aaC \\ C &\to aC|\lambda \end{aligned}$$

7. We can show by induction that if w is a sentential form derived with G, then w^R can be derived in the same number of steps by $\widehat{G}$.

Because w is created with left linear derivations, it must have the form $w = Aw_1$, with $A \in V$ and $w_1 \in T^*$. By the inductive assumption $w^R = w_1^R A$ can be derived via $\widehat{G}$. If we now apply $A \to Bv$, then

$$w \Rightarrow Bvw_1.$$

But $\widehat{G}$ contains the rule $A \to v^R B$, so we can make the derivation

$$\begin{aligned} w^R &\to w_1^R v^R B \\ &= (Bvw_1)^R \end{aligned}$$

completing the inductive step.

10. Split this into two cases: (i) n and m are both even and (ii) n and m are both odd. The solution then falls out easily, with

$$\begin{aligned} S &\to aaS|A \\ A &\to bbA|\lambda \end{aligned}$$

taking care of case (i).

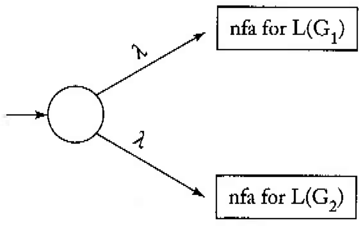
12. (a) First construct a dfa for L. This is straightforward and gives transitions such as

$$\begin{aligned} \delta(q_0, a) &= q_1, \delta(q_0, b) = q_2 \\ \delta(q_1, a) &= q_0, \delta(q_1, b) = q_3 \\ \delta(q_2, a) &= q_3, \delta(q_2, b) = q_0 \\ \delta(q_3, a) &= q_2, \delta(q_3, b) = q_1 \end{aligned}$$

with q_0 the initial and final state. Then the construction of Theorem 3.4 gives the answer

$$\begin{aligned} q_0 &\to aq_1 \,|bq_2|\, \lambda \\ q_1 &\to bq_3|aq_0 \\ q_2 &\to aq_3|bq_0 \\ q_3 &\to aq_2|bq_1 \end{aligned}$$

16. Obviously, S_1 is regular as is S_2. We can show that their union is also regular by constructing the following dfa.



The condition that V_1 and V_2 should be disjoint is essential so that the two nfa's are distinct.

Chapter 4

Section 4.1

2. (a) The construction is straightforward, but tedious. A dfa for $L((a+b)a^*)$ is given by

$$\delta(q_0,a) = q_1, \quad \delta(q_0,b) = q_1, \quad \delta(q_1,a) = q_1, \quad \delta(q_1,b) = q_t,$$

with q_t a trap state and final state q_1. A dfa for $L(baa^*)$ is given by

$$\begin{aligned} &\delta(p_0,a) = p_t, \delta(p_0,b) = p_1, \delta(p_1,a) = p_2, \\ &\delta(p_1,b) = p_t, \delta(p_2,a) = p_2, \delta(p_2,b) = p_t \end{aligned}$$

with final state p_2. From this we find

$$\begin{aligned} &\delta((q_0,p_0),a) = (q_1,p_t), \delta((q_0,p_0),b) = (q_1,p_1), \\ &\delta((q_1,p_1),a) = (q_1,p_2), \delta((q_1,p_2),a) = (q_1,p_2), \end{aligned}$$

etc. When we complete this construction, we see that the only final state is (q_1,p_2) and that $L((a+b)a^*) \cap L(baa^*) = baa^*$.

7. Notice that

$$nor\,(L_1, L_2) = \overline{L_1 \cup L_2}.$$

The result then follows from closure under intersection and complementation.

12. The answer is yes. It can be obtained by starting from the set identity

$$L_2 = \left((L_1 \cup L_2) \cap \overline{L_1}\right) \cup (L_1 \cap L_2).$$

The key observation is that since L_1 is finite, $L_1 \cap L_2$ is finite and therefore regular for all L_2. The rest then follows easily from the known closures under union and complementation.

14. By closure under reversal, L^R is regular. The result then follows from closure under concatenation.

16. Use $L_1 = \Sigma^*$. Then, for any L_2, $L_1 \cup L_2 = \Sigma^*$, which is regular. The given statement would then imply that any L_2 is regular.

18. We can use the following construction. Find all states P such that there is a path from the initial vertex to some element of P, and from that element to a final state. Then make every element of P a final state.

26. Suppose $G_1 = (V_1, T, S_1, P_1)$ and $G_2 = (V_2, T, S_2, P_2)$. Without loss of generality, we can assume that V_1 and V_2 are disjoint. Combine the two grammars and

    (a) Make S the new start symbol and add productions $S \to S_1|S_2$.
    (b) In P_1, replace every production of the form $A \to x$, with $A \in V_1$ and $x \in T^*$, by $A \to xS_2$.
    (c) In P_1, replace every production of the form $A \to x$, with $A \in V_1$, and $x \in T^*$, by $A \to xS_1$.

Section 4.2

1. Since by Example 4.1 $L_1 - L_2$ is regular, there exists a membership algorithm for it.

2. If $L_1 \subseteq L_2$, then $L_1 \cup L_2 = L_2$. Since $L_1 \cup L_2$ is regular and we have an algorithm for set equality, we also have an algorithm for set inclusion.

5. From the dfa for L, construct the dfa for L^R, using the construction suggested in Theorem 4.2. Then use the equality algorithm in Theorem 4.7.

12. Here you need a little trick. If L contains no even length strings, then

$$L \cap L\left((aa + ab + ba + bb)^*\right) = \varnothing.$$

The left side is regular, so we can use Theorem 4.6.

Section 4.3

2. For the dfa for L to process the middle string v requires a walk in the transition graph of length $|v|$. If this is longer than the number of states in the dfa, there must be a cycle labeled y in this walk. But clearly this cycle can be repeated as often as desired without changing the acceptability of a string.

4. (a) Given m, pick $w = a^m b^m a^{2m}$. The string y must then be a^k and the pumped strings will be

$$w_i = a^{m+(i-1)k} b^m a^{2m}.$$

If we take $i \geq 2$ then $m + (i-1)k > m$, then w_i is not in L.

(e) It does not seem easy to apply the pumping lemma directly, so we proceed indirectly. Suppose that L were regular. Then by the closure of regular languages under complementation, $\overline{L}$ would also be regular. But $\overline{L} = \{w : n_a(w) = n_b(w)\}$ which, as is easily shown, is not regular. By contradiction, L is not regular.

5. (a) Take p to be the smallest prime number greater or equal to m and choose $w = a^p$. Now y is a string of a's of length k, so that

$$w_i = a^{p+(i-1)k}.$$

If we take $i - 1 = p$, then $p + (i-1)k = p(k+1)$ is composite and w_{p+1} is not in the language.

8. The proposition is false. As a counterexample, take $L_1 = \{a^n b^m : n \leq m\}$ and $L_2 = \{a^n b^m : n > m\}$, both of which are non-regular. But $L_1 \cup L_2 = L(a^* b^*)$, which is regular.

9. (a) The language is regular. This is most easily seen by splitting the problem into cases such as $l = 0, k = 0, n > 5$, for which one can easily construct regular expressions.

(b) This language is not regular. If we choose $w = aaaaaab^m a^m$, our opponent has several choices. If y consists of only a's, we use $i = 0$ to violate the condition $n > 5$. If the opponent chooses y as consisting of b's, we can then violate the condition $k \leq l$.

11. L is regular. We see this from $L = L_1 \cap L_2^R$ and the known closures for regular languages.

13. (a) The language is regular, since any string that has two consecutive symbols the same is in the language. A regular expression for L is $(a+b)(a+b)^*(aa+bb)(a+b)(a+b)^*$.

(b) The language is not regular. Take $w = (ab)^m aa (ba)^m$. The adversary now has several choices, such as $y = (ab)^k$ or $y = (ab)^k a$. In the first case

$$w_0 = (ab)^{m-k} aa (ba)^m .$$

Since the only possible identification is $ww^R = b^l aab^l$, w_0 is not in L. With the second choice, the length of w_0 is odd, so it cannot be in L either.

15. Take $L_i = a^i b^i, i = 0, 1, \ldots$. For each i, L_i is finite and therefore regular, but the union of all the languages is the non-regular language $L = \{a^n b^n : n \geq 0\}$.

17. No, it is not. As counterexample, take the languages

$$L_i = \{v_i u v_i^R : |v_i| = i\} \cup \{v_i v_i^R : |v_i| < i\}, i = 0, 1, 2, \ldots$$

We claim that the union of all the L_i is the set $\{ww^R\}$. To justify this, take any string $z = ww^R$, with $|w| = n$. If $n \geq i$, then $z \in \{v_i u v_i^R : |v_i| = i\}$ and therefore in L_i. If $n < i$, then $z \in \{v_i v_i^R : |v_i| < i\}$, $i = \{0, 1, 2, \ldots\}$ and so also in L_i. Consequently, z is in the union of all the L_i.

Conversely, take any string z of length m that is in all of the L_i. If we take i greater than m, z cannot be in $\{v_i u v_i^R : |v_i| = i\}$ because it is not long enough. It must therefore be in $\{v_i v_i^R : |v_i| < i\}$, so that it has the form ww^R.

As the final step we must show that for each i, L_i is regular. This follows from the fact that for each i there are only a finite number of substrings v_i.

Chapter 5

Section 5.1

4. It is quite obvious that any string generated by this grammar has the same number of a's as b's. To show that the prefix condition $n_a(v) \geq n_b(v)$ holds, we carry out an induction on the length of the derivation. Suppose that for every sentential form derived from S in n steps this condition holds. To get a sentential form in $n + 1$ steps, we can apply $S \rightarrow \lambda$ or $S \rightarrow SS$. Since neither of these changes the number of a's and b's or the location of those already there, the prefix condition continues to hold. Alternatively, we apply $S \rightarrow aSb$. This adds an extra a and an extra b, but since the added a is to the left of the added b, the prefix condition will still be satisfied. Thus, if the prefix condition holds after n steps, it will still hold after $n+1$ steps. Obviously, the prefix condition holds after one step, so we have a basis and the induction succeeds.

7. (a) First, solve the case $n = m + 3$. Then add more b's. This can be done by

$$\begin{aligned} S &\rightarrow aaaA \\ A &\rightarrow aAb|B \\ B &\rightarrow Bb|\lambda \end{aligned}$$

But this is incomplete since it creates at least three a's. To take care of the cases $n = 0, 1, 2$, we add

$$S \rightarrow \lambda\, |aA|\, aaA$$

(d) This has an unexpectedly simple solution

$$S \rightarrow aSbb\, |aSbbb|\, \lambda.$$

These productions nondeterministically produce either bb or bbb for each generated a.

8. (a) For the first case $n = m$ and k is arbitrary. This can be achieved by

$$\begin{aligned} S_1 &= AC \\ A &\rightarrow aAb|\lambda \\ C &\rightarrow Cc|\lambda \end{aligned}$$

In the second case, n is arbitrary and $m \le k$. Here we use

$$\begin{aligned} S_2 &\rightarrow BD \\ B &\rightarrow aB|\lambda \\ D &\rightarrow bDc|E \\ E &\rightarrow Ec|\lambda. \end{aligned}$$

Finally, we start productions with $S \rightarrow S_1|S_2$.

(e) Split the problem into two cases: $n = k + m$ and $m = k + n$. The first case is solved by

$$\begin{aligned} S &\rightarrow aSc\, |S_1|\, \lambda \\ S_1 &\rightarrow aS_1b|\lambda. \end{aligned}$$

12. (a) If S derives L, then $S_1 \rightarrow SS$ derives L^2.

15. It is normally not possible to use a grammar for L directly to get a grammar for $\overline{L}$, so we need another, hopefully recursive description for

$\overline{L}$. This is a little hard to see here. One obvious subset of $\overline{L}$ contains the strings of odd length, but this is not all.

Suppose we have an even length string that is not of the form ww^R. Working from the center to the left and to the right simultaneously, compare corresponding symbols. While some part around the center can be of the form ww^R, at some point we get an a on the left and a b in the corresponding place on the right, or vice versa. The string must therefore be of the form $uaww^Rbv$ or $ubww^Rav$ with $|u| = |v|$. Once we see this, we can then construct grammars for these types of strings. One solution is

$$\begin{aligned} &S \to ASA|B \\ &A \to a|b \\ &B \to bCa|aCb \\ &C \to aCa\,|bCb|\,\lambda. \end{aligned}$$

The first two productions generate the u and v, the third the two disagreeing symbols, and the last the innermost palindrome.

19. The only possible derivations start with

$$S \Rightarrow aaB \Rightarrow aaAa \Rightarrow aabBba \Rightarrow aabAaba.$$

But this sentential form has the suffix aba so it cannot possibly lead to the sentence $aabbabba$.

22. $E \to E + E\,|E.E|\,E^*\,|(E)|\,\lambda|\varnothing.$

Section 5.2

2. A solution is

$$S \to aA, A \to aAB|b, B \to b.$$

Note that the more obvious grammar

$$\begin{aligned} &S \to aS_1B \\ &S_1 \to aS_1B|\lambda \\ &B \to b \end{aligned}$$

is not an s-grammar.

6. There are two leftmost derivations for $w = aab$.

$$\begin{aligned} &S \Rightarrow aaB \Rightarrow aab \\ &S \Rightarrow AB \Rightarrow AaB \Rightarrow aaB \Rightarrow aab. \end{aligned}$$

9. From the dfa for a regular language we can get a regular grammar by the method of Theorem 3.4. The grammar is an s-grammar except for $q_f \to \lambda$. But this rule does not create any ambiguity. Since the dfa never has a choice, there is never any choice in the production that can be applied.

14. Ambiguity of the grammar is obvious from the derivations

$$S \Rightarrow aSb \Rightarrow ab$$
$$S \Rightarrow SS \Rightarrow abS \Rightarrow ab.$$

An equivalent unambiguous grammar is

$$S \to A|\lambda$$
$$A \to aAb\,|ab|\,AA.$$

It is not easy to see that this grammar is unambiguous. To make it plausible, consider the two typical situations, $w = aabb$, which can only be derived by starting with $A \to aAb$, and $w = abab$, which can only be derived starting with $A \to AA$. More complicated strings are built from these two situations, so they can be parsed only in one way.

20. Solution:

$$S \to aA|aAA$$
$$A \to bAb|bb.$$

Chapter 6

Section 6.1

3. Use the rule in Theorem 6.1 to substitute for B in the first grammar. Then B becomes useless and the associated productions can be removed. By Theorems 6.1 and 6.2 the two grammars are equivalent.

8. The only nullable variable is A, so removing λ-productions gives

$$S \to aA\,|a|\,aBB$$
$$A \to aaA|aa$$
$$B \to bC|bbC$$
$$C \to B.$$

$C \to B$ is the only unit-production and removing it results in

$$S \to aA\,|a|\,aBB$$
$$A \to aaA|aa$$
$$B \to bC|bbC$$
$$C \to bC|bbC.$$

Finally, B and C are useless, so we get

$$S \rightarrow aA|a$$
$$A \rightarrow aaA|aa.$$

The language generated by this grammar is $L\left((aa)^* a\right)$.

14. An example is

$$S \rightarrow aA$$
$$A \rightarrow BB$$
$$B \rightarrow aBb|\lambda.$$

When we remove λ-productions we get

$$S \rightarrow aA|a$$
$$A \rightarrow BB|B$$
$$B \rightarrow aBb|ab.$$

16. This is obvious since the removal of useless productions never adds anything to the grammar.

21. The grammar $S \rightarrow aA$; $A \rightarrow a$ does not have any useless productions, any unit productions, or any λ-productions. But it is not minimal since $S \rightarrow aa$ is an equivalent grammar.

Section 6.2

5. First we must eliminate λ-productions. This gives

$$S \rightarrow AB\,|B|\,aB$$
$$A \rightarrow aab$$
$$B \rightarrow bbA|bb.$$

This has introduced a unit-production, which is not acceptable in the construction of Theorem 6.6. Removal of this unit-production is easy.

$$S \rightarrow AB\,|bbA|\,aB|bb$$
$$A \rightarrow aab$$
$$B \rightarrow bbA|bb.$$

We can now apply the construction and get

$$S \rightarrow AB\,|V_bV_bA|\,V_aB|V_bV_b$$
$$A \rightarrow V_aV_bV_b$$
$$B \rightarrow V_bV_bA|V_bV_b$$

and

$$\begin{aligned} S &\to AB\,|V_cA|\,V_aB|V_bV_b \\ A &\to V_dV_b \\ B &\to V_cA|V_bV_b \\ V_c &\to V_bV_b \\ V_d &\to V_aV_b \\ V_a &\to a \\ V_b &\to b. \end{aligned}$$

8. Consider the general form for a production in a linear grammar

$$A \to a_1a_2...a_nBb_1b_2...b_m.$$

Introduce a new variable V_1 with the productions

$$V_1 \to a_2...a_nBb_1b_2...b_m$$

and

$$A \to a_1V_1.$$

Continue this process, introducing V_2 and

$$V_2 \to a_3...a_nBb_1b_2...b_m$$

and so on, until no terminals remain on the left. Then use a similar process to remove terminals on the right.

9. This normal form can be reached easily from CNF. Productions of the form $A \to BC$ are permitted since $a = \lambda$ is possible. For $A \to a$, create new variables V_1, V_2 and productions $A \to aV_1V_2$, $V_1 \to \lambda$, $V_2 \to \lambda$.

12. Solutions: $S \to aV_b\,|aS|\,aV_aS$, $V_a \to a$, $V_b \to b$.

15. Only $A \to bABC$ is not in the required form, so we introduce $A \to bAV$ and $V \to BC$. The latter is not in correct form, but after substituting for B, we have

$$\begin{aligned} S &\to aSA \\ A &\to bAV \\ V &\to bC \\ C &\to aBC. \end{aligned}$$

Section 6.3

2. Since aab is a prefix of the string in Example 6.11, we can use the V_{ij} computed there. Since $S \in V_{13}$, the string aab is in the language generated by the grammar and can therefore be parsed.

For parsing, we determine the productions that were used in justifying $S \in V_{13}$:

$$\begin{aligned}
&S \in V_{13} \text{ because } S \to AB, \text{ with } A \in V_{11} \text{ and } B \in V_{23}\\
&A \in V_{11} \text{ because } A \to a\\
&B \in V_{23} \text{ because } B \to AB, \text{ with } A \in V_{22}, B \in V_{33}\\
&A \in V_{22} \text{ because } A \to a\\
&B \in V_{33} \text{ because } B \to b.
\end{aligned}$$

This shows all the productions needed to justify membership; these can then be used in the parsing

$$S \Rightarrow AB \Rightarrow aB \Rightarrow aAB \Rightarrow aaB \Rightarrow aab.$$

Chapter 7

Section 7.1

2. The key to the argument is the switch from q_0 to q_1, which is done nondeterministically and need not happen in the middle of the string. However, if a switch is made at some other point or if the input is not of the form ww^R, an accepting configuration cannot be reached. Suppose the content of the stack at the time of the switch is $x_1x_2...x_kz$. To accept a string we must get to the configuration (q_1, λ, z). By examining the transition function, we see that we can get to this configuration only if at this point the unread part of the input is $x_1x_2...x_k$, that is, if the original input is of the form ww^R and the switch was made exactly in the middle of the input string.

4. (a) The solution is obtained by letting each a put two markers on the stack, while each b consumes one. Solution:

$$\begin{aligned}
\delta(q_0, \lambda, z) &= \{(q_f, z)\}\\
\delta(q_0, a, z) &= \{(q_1, 11z)\}\\
\delta(q_0, a, 1) &= \{(q_1, 111)\}\\
\delta(q_1, b, 1) &= \{(q_1, \lambda)\}\\
\delta(q_1, \lambda, z) &= \{(q_f, z)\}.
\end{aligned}$$

(f) Here we use nondeterminism to generate one, two, or three tokens by

$$\delta(q_0, a, z) = \{(q_1, 1z), (q_1, 11z), (q_1, 111z)\}$$

and

$$\delta(q_0, a, z) = \{(q_1, 11), (q_{10}, 111), (q_1, 1111)\}.$$

The rest of the solution is then essentially the same as 4(a).

9. This is a pda that makes no use of the stack, so that is, in effect, a finite accepter. The state transitions can then be taken directly from the pda, to give

$$\begin{aligned}\delta(q_0, a) &= q_1\\ \delta(q_0, b) &= q_0\\ \delta(q_1, a) &= q_1\\ \delta(q_1, b) &= q_0\end{aligned}$$

11. Trace through the process, taking one path at a time. The transition from q_0 to q_2 can be made with a single a. The alternative path requires one a, followed by one or more b's, terminated by an a. These are the only choices. The pda therefore accepts the language

$$L = \{a\} \cup L(abb^*a).$$

14. Here we are not allowed enough states to track the switch from a's to b's and back. To overcome this, we put a symbol in the stack that remembers where in the sequence we are. For example, a solution is

$$\begin{aligned}\delta(q_0, a, z) &= \{(q_0, 1)\},\\ \delta(q_0, a, 1) &= \{(q_0, 1)\},\\ \delta(q_0, b, 1) &= \{(q_0, 2)\},\\ \delta(q_0, a, 2) &= \{(q_0, 2)\},\\ \delta(q_0, \lambda, 2) &= \{(q_f, 2)\}.\end{aligned}$$

We have only two states, the initial state q_0 and the accepting state q_f. What would normally be tracked by different states is now tracked by the symbol in the stack.

16. Here we use internal states to remember symbols to be put on the stack. For example,

$$\delta(q_i, a, b) = \{(q_j, cde)\}$$

is replaced by

$$\begin{aligned}\delta(q_i, a, b) &= \{(q_{jc}, de)\}\\ \delta(q_{jc}, \lambda, d) &= \{(q_j, cd)\}.\end{aligned}$$

Since δ can have only a finite number of elements and each can only add a finite amount of information to the stack, this construction can be carried out for any pda.

Section 7.2

3. You can follow the construction of Theorem 7.1 or you can notice that the language is $\{a^{n+2}b^{2n+1} : n \geq 0\}$. With the latter observation we get a solution

$$
\begin{aligned}
\delta(q_0, a, z) &= \{(q_1, z)\} \\
\delta(q_1, a, z) &= \{(q_2, z)\} \\
\delta(q_2, a, z) &= \{(q_2, 11z)\} \\
\delta(q_2, a, 1) &= \{(q_2, 111)\} \\
\delta(q_2, b, 1) &= \{(q_3, 1)\} \\
\delta(q_3, b, 1) &= \{(q_3, \lambda)\} \\
\delta(q_3, \lambda, z) &= \{(q_f, z)\}
\end{aligned}
$$

where q_0 is the initial state and q_f is the final state.

4. First convert the grammar into Griebach normal form, giving $S \rightarrow aSSS$; $S \rightarrow aB$; $B \rightarrow b$. Then follow the construction of Theorem 7.1.

$$
\begin{aligned}
\delta(q_0, \lambda, z) &= \{(q_1, Sz)\} \\
\delta(q_1, a, S) &= \{(q_1, SSS), (q_1, B)\} \\
\delta(q_1, b, B) &= \{(q_1, \lambda)\} \\
\delta(q_1, \lambda, z) &= \{(q_f, z)\}.
\end{aligned}
$$

7. From Theorem 7.2, given any npda, we can construct an equivalent context-free grammar. From that grammar we can then construct an equivalent three-state npda, using Theorem 7.1. Because of the transitivity of equivalence, the original and the final npda's are also equivalent.

9. We first obtain a grammar in Greibach normal form for L, for example $S \rightarrow aSB|b, B \rightarrow b$. Next, we apply the construction in Theorem 7.1 to get an npda with three states, q_0, q_1, q_f. The state q_1 can be eliminated if we use a special stack symbol z_1 to mark it. A complete solution is

$$
\begin{aligned}
\delta(q_0, \lambda, z) &= \{(q_0, Sz_1)\} \\
\delta(q_0, a, S) &= \{(q_0, SB)\} \\
\delta(q_0, b, S) &= \{(q_0, \lambda)\} \\
\delta(q_0, b, B) &= \{(q_0, \lambda)\} \\
\delta(q_0, \lambda, z_1) &= \{(q_f, \lambda)\}.
\end{aligned}
$$

11. There must be at least one a to get started. After that, $\delta(q_0, a, A) = \{(q_0, A)\}$ simply reads a's without changing the stack. Finally, when

the first b is encountered, the pda goes into state q_1, from which it can only make a λ-transition to the final state. Therefore, a string will be accepted if and only if it consists of one or more a's, followed by a single b.

Section 7.3

4. At first glance, this may seem to be a nondeterministic language, since the prefix a calls for two different types of suffixes. Nevertheless, the language is deterministic, as we can construct a dpda. This dpda, goes into a final state when the first input symbol is an a. If more symbols follow, it goes out of this state and then accepts $a^n b^n$. Complete solution:

$$\begin{aligned}
\delta(q_0, a, z) &= \{(q_3, 1z)\} \\
\delta(q_3, a, 1) &= \{(q_1, 11)\} \\
\delta(q_1, a, 1) &= \{(q_1, 11)\} \\
\delta(q_1, b, 1) &= \{(q_1, \lambda)\} \\
\delta(q_1, \lambda, z) &= \{(q_2, z)\}
\end{aligned}$$

where $F = \{q_2, q_3\}$.

9. The solution is straightforward. Put a's and b's on the stack. The c signals the switch from saving to matching, so everything can be done deterministically.

11. There are two states, the initial, non-accepting state q_0 and the final state q_1. The pda will be in state q_1 unless a z is on top of the stack. When this happens, the pda will switch states to q_0. The rest is essentially the same as Example 7.3. Thus we have $\delta(q_0, a, z) = \{(q_1, 0z)\}, \delta(q_1, a, 0) = \{(q_1, 00)\}$, etc. with $\delta(q_1, \lambda, z) = \{(q_0, z)\}$. When you write this all out, you will see that the pda is deterministic.

15. This is obvious since every regular language can be accepted by a dfa and such a dfa is a dpda with an unused stack.

16. The basic idea here is to combine a dpda with a dfa along the lines of the construction in Theorem 4.1, with the stack handled as it is for L_1. It should not be too hard to see that the result is a dpda.

Section 7.4

2. Consider the strings $aabb$ and $aabbbbaa$. In the first case, the derivation must start with $S \Rightarrow aSB$, while in the second $S \Rightarrow SS$ is the necessary first step. But if we see only the first four symbols, we cannot decide which case applies. The grammar is therefore not in $LL(4)$. Since

similar examples can be made for arbitrarily long strings, the grammar is not $LL(k)$ for any k.

4. Look at the first three symbols. If they are aaa, aab, or aba, then the string can only be in $L(a^*ba)$. If the first three symbols are abb, then any parsable string must be in $L(abbb^*)$. For each case, we can find an LL grammar and the two can be combined in an obvious fashion. A solution is

$$\begin{aligned} S &\to S_1|S_2 \\ S_1 &\to aS_1|ba \\ S_2 &\to abbB \\ B &\to bB|\lambda. \end{aligned}$$

Looking at the first three symbols tells us if $S \Rightarrow S_1$ or $S \Rightarrow S_2$ is necessary. The grammar is therefore $LL(3)$.

7. For a deterministic CFL there exists a dpda. When this dpda is converted into a grammar, the grammar is unambiguous.

9. (a)

$$\begin{aligned} S &\to aSc\,|S_1|\,\lambda \\ S_1 &\to bS_1c|\lambda. \end{aligned}$$

This is almost an s-grammar. As long as the currently scanned symbol is a, we must apply $S \to aSc$, if it is b, we must use $S \to S_1$, if it is c, we can only use $S \to \lambda$. The grammar is $LL(1)$.

Chapter 8

Section 8.1

3. Take $w = a^m b^m b^m a^m a^m b^m$. The adversary now has several choices that have to be considered. If, for example, $v = a^k$ and $y = a^l$, with v and y located in the prefix a^m, then

$$w_0 = a^{m-k-l} b^m b^m a^m a^m b^m,$$

which is not in L. There are a number of other possible choices, but in all cases the string can be pumped out of the language.

7. (a) Use the pumping lemma. Given m, pick $w = a^{m^2} b^m$. The only choice of v and y that needs any serious examination is $v = a^k$ and

$y = b^l$, with k and l non-zero. Suppose that $l = 1$. Then choose $i = 2$, so that w_2 has $m^2 + k$ a's and $m + 1$ b's. But

$$\begin{aligned}(m+1)^2 &= m^2 + 2m + 1 \\ &> m^2 + k.\end{aligned}$$

Since w_2 is not in the language, the language cannot be context-free. Similar arguments hold a fortiori for $l > 1$.

(f) Given m, choose $w = a^m b^{m+1} c^{m+2}$, which is easily pumped out of the language.

8. (b) The language is not context-free. Use the pumping lemma with $w = a^m b^m a^m b^m$ and examine various choices of v and y.

10. Perhaps surprisingly, this language is context-free. Construct an npda that counts to some value k (by putting k tokens on the stack) and remembers the k-th symbol. It then examines the k-th symbol in w_2. If this does not match the remembered symbol, the string is accepted. If $w \in L$ there must be some k for which this happens. The npda chooses the k nondeterministically.

12. Use the pumping lemma for linear languages. With a given m, choose $w = a^m b^{2m} a^m$. Now v and y are entirely made of a's, so w is easily pumped out of the language.

15. The language is not linear. With the pumping lemma, use

$$w = (\ldots(a)\ldots) + (\ldots(a)\ldots)$$

where $(\ldots($ and $)\ldots)$ stand for m left or right parentheses, respectively. If $|u| \geq 1$, we can easily pump so that for some prefix v, $n_((v) < n_) (v)$ which results in an improper expression. Similar arguments hold for other decompositions.

20. Use $w = a^{pq}$, where p and q are primes such that $p > m$ and $q > m$. If $|vy| = k$, then

$$|w_{i+1}| = pq + ik.$$

If we choose $i = pq$, then

$$w_{i+1} = a^{pq(1+k)},$$

which is not in the language.

Section 8.2

1. The complement is context-free. The complement involves two cases: $n_a(w) \neq n_b(w)$ and $n_a(w) \neq n_c(w)$. These in turn can be broken into $n_a(w) > n_b(w)$, $n_a(w) > n_c(w)$, $n_a(w) < n_b(w)$, and $n_a(w) < n_c(w)$. Each of these is context-free as can be shown by construction of a CFG. The full language is then the union of these four cases and by closure under union is context-free.

5. Given a context-free grammar G, construct a context-free grammar $\widehat{G}$ by replacing every production $A \to x$ by $A \to x^R$. We can then show by an induction on the number of steps in a derivation that if w is a sentential form for G then w^R is a sentential form for $\widehat{G}$.

9. Given two linear grammars $G_1 = (V_1, T, S_1, P_1)$ and $G_2 = (V_2, T, S_2, P_2)$ with $V_1 \cap V_2 = \varnothing$, form the combined grammar $\widehat{G} = (V_1 \cup V_2, T, S, P_1 \cup P_2 \cup S \to S_1 | S_2)$. Then $\widehat{G}$ is linear and $L\left(\widehat{G}\right) = L(G_1) \cup L(G_2)$.

   To show that linear languages are not closed under concatenation, take the linear language $L = \{a^n b^n : n \geq 1\}$. The language L^2 is not linear as can be shown by an application of the pumping lemma.

13. Let $G_1 = (V_1, T, S_1, P_1)$ be a linear grammar for L_1 and let $G_2 = (V_2, T, S_2, P_2)$ be a left-linear grammar for L_2. Construct a grammar $\widehat{G}_2$ from G_2 by replacing every production of the form $V \to x, x \in T^*$ with $V \to S_1 x$. Combine grammars G_1 and $\widehat{G}_2$, choosing S_2 as a start symbol. It is then easily shown that in this grammar

$$S_2 \overset{*}{\Rightarrow} S_1 w \overset{*}{\Rightarrow} uw$$

   if and only if $u \in L_1$ and $w \in L_2$.

15. The languages $L_1 = \{a^n b^n c^m\}$ and $L_2 = \{a^n b^m c^m\}$ are both unambiguous. But their intersection is not even context-free.

21. $\lambda \in L(G)$ if and only if S is nullable.

Chapter 9

Section 9.1

2. A three-state solution that scans the entire input is

$$\begin{aligned} \delta(q_0, a) &= (q_1, a, R) \\ \delta(q_1, a) = \delta(q_1, b) &= (q_1, a, R) \\ \delta(q_1, \square) &= (q_2, \square, R) \end{aligned}$$

   with $F = \{q_2\}$.

It is also possible to get a two-state solution by just examining the first symbol and ignoring the rest of the input, for example,

$$\delta(q_0, a) = (q_2, a, R).$$

7. (a)

$$\begin{aligned}\delta(q_0, a) &= (q_1, a, R)\\ \delta(q_1, b) &= (q_2, b, R)\\ \delta(q_2, a) &= (q_2, a, R)\\ \delta(q_2, b) &= (q_3, b, R)\end{aligned}$$

with $F = \{q_3\}$.

(b)

$$\begin{aligned}\delta(q_0, a) = \delta(q_0, b) &= (q_1, \square, R)\\ \delta(q_0, \square) &= (q_2, \square, R)\\ \delta(q_1, a) = \delta(q_1, b) &= (q_0, \square, R)\end{aligned}$$

with $F = \{q_2\}$.

10. The solution is conceptually simple, but tedious to write out in detail. The general scheme looks something like this:

(i) Place a marker symbol c at each end of the string.

(ii) Replace the two-symbol combination ca on the left by ac and the two-symbol combination ac on the right by ca. Repeat until the two c's meet in the middle of the string.

(iii) Remove one of the c's and move the rest of the string to fill the gap.

Obviously this is a long job, but it is typical of the cumbersome ways in which Turing machines often do simple things.

12. We cannot just search in one direction since we don't know when to stop. We must proceed in a back-and-forth fashion, placing markers at the right and left boundaries of the searched region and moving the markers outward.

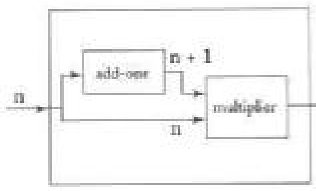
19. If the final state set F contains more than one element, introduce a new final state q_f and the transitions

$$\delta(q, a) = (q_f, a, R)$$

for all $q \in F$ and $a \in \Gamma$.

Section 9.2

3. (a) We can think of the machine as constituted of two main parts, an *add-one* machine that just adds one to the input, and a multiplier that multiplies two numbers. Schematically they are combined in a simple fashion.



5. (c) First, split the input into two equal parts. This can be done as suggested in Exercise 10, Section 9.1. Then compare the two parts, symbol by corresponding symbol until a mismatch is found.

8. A solution:

$$\begin{aligned}\delta(q_0, a) &= (q_i, a, R),\\ \delta(q_0, c) &= (q_0, c, R) \text{ for all } c \in \Sigma - \{a\},\\ \delta(q_0, \square) &= (q_j, \square, R).\end{aligned}$$

The state q_0 is any state in which the *searchright* instruction may be applied.

Section 9.3

2. We have ignored the fact that a Turing machine, as defined so far, is deterministic, while a pda can be non-deterministic. Therefore, we cannot yet claim that Turing machines are more powerful than a pushdown automata.

Chapter 10

Section 10.1

4. (a) The machine has a transition function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

with the restriction that for all transitions $\delta(q_i, a) = (q_j, b, L \text{ or } R)$, the condition $a = b$ must hold.

(b) To simulate $\delta(q_i, a) = (q_j, b, L)$ with $a \neq b$ of the standard machine, we introduce new transitions $\delta(q_i, a) = (q_{jL}, b, S)$ and $\delta(q_{jL}, b) = (q_j, b, L)$ for all $c \in \Gamma$, and so on.

6. We introduce a pseudo-blank B. Whenever the original machine wants to write $\square$, the new machine writes B. Then, for each $\delta(q_i, \square) = (q_j, b, L)$ we add $\delta(q_i, B) = (q_j, b, L)$, and so on. Of course, the original transition $\delta(q_i, \square) = (q_j, b, L)$ must be retained to handle blanks that are originally on the tape.

9. This does not limit the power of the machine. For each symbol $a \in \Gamma$, we introduce a pseudo-symbol, say A. Whenever we need to preserve this a, we first write A, then return to the cell in question to replace A by a.

11. We replace

$$\delta(q_i, \{a, b\}) = (q_j, c, R)$$

by

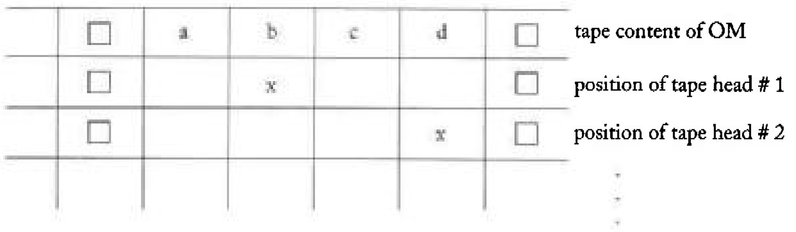
$$\delta(q_i, d) = (q_j, c, R)$$

for all $d \in \Gamma - \{a, b\}$.

Section 10.2

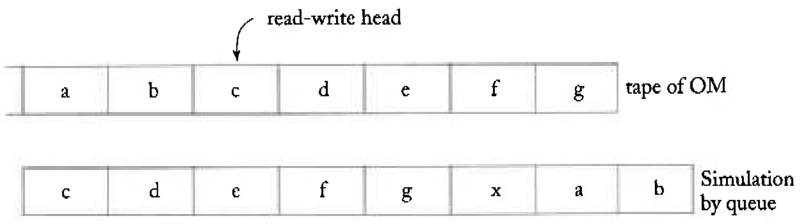
1. For the formal definition use $\Gamma_T = \Gamma \times \Gamma \times ... \times \Gamma$ and $\delta : Q \times \Gamma_T \rightarrow Q \times \Gamma_T \times \{L, R\}^m$, where m is the number of read-write heads. One issue to consider is what happens when two read-write heads are on the same cell. The formal definition must provide for the resolution of possible conflicts.

To simulate the original machine (OM) by a standard Turing machine (SM), we let SM have $m + 1$ tracks. On one track we will keep the tape contents of the OM, while the other m tracks are used to show the position of OM's tape heads.



SM will simulate each move of OM by scanning and updating its active area.

4. This exercise shows that a queue machine is equivalent to a standard Turing machine and that therefore a queue is a more powerful storage device than a stack. To simulate a standard TM by a queue machine, we can, for example, keep the right side of the OM in the front of the queue, the left side in the back.



A right move is easy as we just remove the front symbol in the queue and place something in the back. A left move, however, goes against the grain, so the queue contents have to be circulated several times to get everything in the right place. It helps to use additional markers Y and Z to denote boundaries. For example, to simulate

$$\delta(q_i, c) = (q_j, z, L)$$

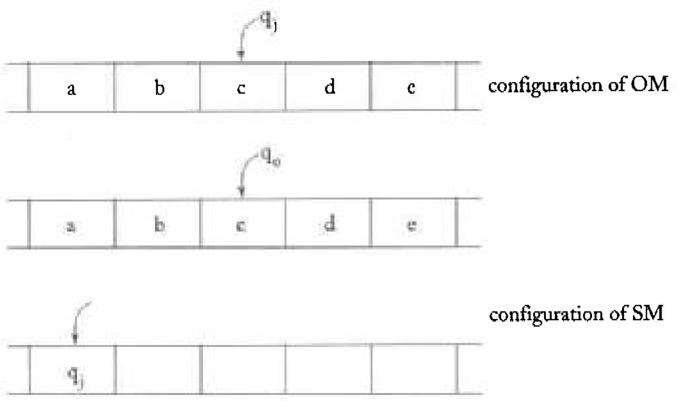
we carry out the following steps.

(i) Remove c from the front and add zY to the back.

(ii) Circulate contents to get $bzYdefgXa$.

(iii) Add Z to the back, then circulate, discarding Y and Z as they come to the front.

8. We need just two tapes, one that mirrors the tape of the OM, the second that stores the state of the OM.



SM needs only two states: an accepting and a non-accepting state.

Section 10.3

3. (i) Start at the left of the input. Remember the symbol by putting the machine in the appropriate state. Then replace it with X.

(ii) Move the read-write head to the right, stopping (nondeterministically) at the center of the input.

(iii) Compare the symbol there with the remembered one. If they match, write Y in the cell. If they don't match, reject input.

(iv) With the center of the input marked with Y, we can now proceed deterministically, alternatively moving left and right, comparing symbols.

For a completely deterministic solution, we first find the center of the input (e.g. by putting markers at each end, and moving them inwards until they meet).

6. Nondeterministically choose a value for n. Determine if the length of the input is a multiple of n. If it is, accept. If $a^n \in L$, then there is some n for which this works.

Section 10.4

3. An algorithm, in outline, is as follows.

(i) Start with a copy of the preceding string.

(ii) Find the rightmost 0. Change it to a 1. Then change all the 1's to the right of this to 0's.

(iii) If there are no 0's, change all 1's to 0's and add a 1 on the left.

(iv) Repeat from step (i).

8. Let $S_1 = \{s_1, s_2, ...\}$ and $S_2 = \{t_1, t_2, ...\}$ Then their union can be enumerated by

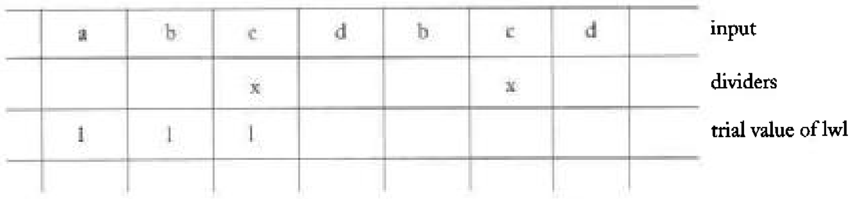
$$S_1 \cup S_2 = \{s_1, t_1, s_2, t_2, ...\}.$$

If some $s_i = t_j$, we list it only once. The union of the two sets is therefore countable. For $S_1 \times S_2$, use the ordering in Figure 10.17.

Section 10.5

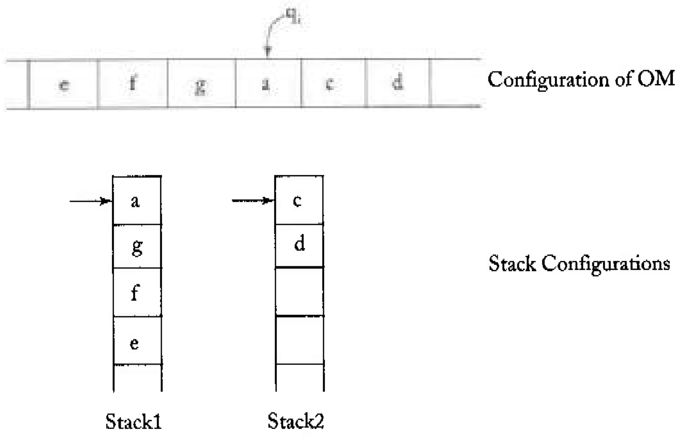
2. First, divide the input by two and move result to one part of tape. This free space initially occupied by the input. This space can then be used to store successive divisors.

4. (e) Use a three-track machine as shown below. On the third track, we keep the current trial value for $|w|$. On the second track, we place dividers every $|w|$ cells. We then compare the cell contents between the markers.



6. Use Exercise 16, Section 6.2 to find a grammar in two-standard form. Then use the construction in Theorem 7.1. The pda we get from this consumes one input symbol on every move and never increases the stack contents by more than one symbol each time.

7. Example:



Stack1 contains the symbol under the read-write head of the *OM* and everything on the left. *Stack2* contains all the information to the right of the read-write head. Left and right moves of the *OM* are easily simulated. For example, $\delta(q_i, a) = (q_j, b, L)$ can be simulated by popping the a off *Stack1* and putting a b on *Stack2*.

Chapter 11

Section 11.1

2. We know that the union of two countable sets is countable and that the set of all recursively enumerable languages is countable. If the set of

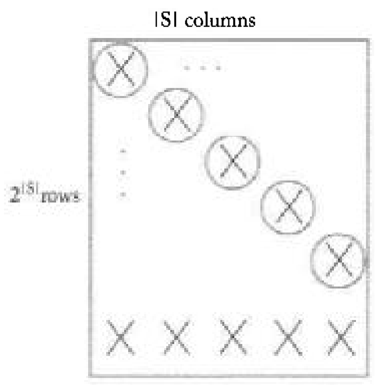
all languages that are not recursively enumerable were also countable, then the set of all languages would be countable. But this is not the case, as we know.

6. Let L_1 and L_2 be two recursively enumerable languages and M_1 and M_2 be the respective Turing machines that accept these two languages. When represented with an input w, we nondeterministically choose M_1 or M_2 to process w. The result is a Turing machine that accepts $L_1 \cup L_2$.

11. A context-free language is recursive, so by Theorem 11.4 its complement is also recursive. Note, however, that the complement is not necessarily context-free.

14. For any given $w \in L^+$, consider all splits $w = w_1 w_2 ... w_m$. For each split, determine whether or not $w_i \in L$. Since for each w there are only a finite number of splits, we can decide whether or not $w \in L^+$.

18. The argument attempting to show by diagonalization that 2^S is not countable for finite S fails because the table in Figure 11.2 is not square, having $|2^S|$ rows and $|S|$ columns.



When we diagonalize, the result on the diagonal could be in one of the rows below.

Section 11.2

1. Look at a typical derivation:

$$S \overset{*}{\Rightarrow} aS_1bB \Rightarrow aaS_1bbB \overset{*}{\Rightarrow} a^nS_1b^nB \Rightarrow a^{n+1}b^{n-1}B \Rightarrow a^{n+1}b^{n+1}B \Rightarrow ...$$

From this it is not hard to conjecture that the grammar derives

$$L = \{a^{n+1}b^{n+k}, n \geq 1, k = -1, 1, 3, ...\}.$$

3. Formally, the grammar can be described by $G = (V, S, T, P)$, with $S \subseteq (V \cup T)^+$ and

$$L(G) = \{x \in T^* : s \Rightarrow_G x \text{ for any } s \in S\}.$$

The unrestricted grammars in Definition 11.3 are equivalent to this extension because to any given unrestricted grammar we can always add starting rules $S_0 \to s_i$ for all $s_i \in S$.

7. To get this form for unrestricted grammars, insert dummy variables on the right whenever $|u| > |v|$. For example,

$$AB \to C$$

can be replaced by

$$AB \to CD$$
$$D \to \lambda.$$

The equivalence argument is straightforward.

Section 11.3

1. (c) Working with context-sensitive grammars is not always easy. The idea of a messenger, introduced in Example 11.2, is often useful.

In this problem, the first step is to create the sentential form $a^n Bc^n D$. The variables B and D will act as markers and messengers to assure that the correct number of b's and d's are created in the right places. The first part is achieved easily with the productions

$$S \to aAcD|aBcD$$
$$A \to aAc|aBc.$$

In the next step, the B travels to the right to meet the D, by

$$Bc \to cB$$
$$Bb \to bB.$$

When that happens, we can create one d and a return messenger that will put the b in the right place and stop.

$$BD \to Ed$$
$$cE \to Ec$$
$$bE \to Eb$$
$$aE \to ab.$$

Alternatively, we create a d plus a marker D, with a different messenger that creates a b, but keeps the process going:

$$\begin{aligned} BD &\to FDd \\ cF &\to Fc \\ bF &\to Fb \\ aF &\to abB. \end{aligned}$$

4. The easiest argument is from an lba. Suppose that a language is context-sensitive. Then there exists an lba M that accepts it. Given w, we first rewrite it as w^R, then apply M to it. Because $L^R = \{w : w^R \in L\}$, M accepts w^R if and only if $w \in L^R$. The machine that reverses a string and applies M is an lba. Therefore L^R is context-sensitive.

6. We can argue from an lba. Clearly, there is an lba that can recognize any string of the form wuw. Just start at opposite ends and compare symbols until you get a match. Since there is an lba, the language is context-sensitive and a context-sensitive grammar must exist.

Chapter 12

Section 12.1

3. Given M and w, modify M to get $\widehat{M}$, which halts if and only if a special symbol, say an introduced symbol $\#$, is written. We can do this by changing the halting configurations of M so that every one writes $\#$, then stops. Thus, M halts implies the $\widehat{M}$ writes $\#$, and $\widehat{M}$ writes $\#$ implies that M halts. Thus, if we have an algorithm that tells us whether or not a specified symbol a is ever written, we apply it to $\widehat{M}$ with $a = \#$. This would solve the halting problem.

7. Given (M, w) modify M to $\widehat{M}$ so that (M, w) halts if and only if $\widehat{M}$ accepts some simple language, say $\{a\}$. This can be done by M first checking the input and remembering whether the input was a. Then M carries out its normal computations. When it halts, check if the input was a. Accept if so, reject otherwise. Therefore $\widehat{M}$ accepts $\{a\}$ if and only if M halts. Now construct a simple Turing machine, say M_1, that accepts a. If we had an algorithm that checks for the equality of two languages, we could use it to see if $L\left(\widehat{M}\right) = L(M_1)$. If $L\left(\widehat{M}\right) = L(M_1)$ then (M, w) halts. If $L\left(\widehat{M}\right) \neq L(M_1)$ then (M, w) does not halt and we have a solution to the halting problem.

10. Given (M, w) we modify M so that it always halts in the configuration $q_f w$. If the given problem was decidable, we could apply the supposed algorithm to the modified machine, with configurations $q_0 w$ and $q_f w$. This would give us a solution of the halting problem.

13. Take a universal Turing machine and let it simulate computations on an empty tape. Whenever the simulated computations halt, accept the Turing machine being simulated. The universal Turing machine is therefore an accepter for all Turing machines that halt when applied to a blank tape. The set is therefore recursively enumerable.

Suppose now the set were recursive. There would then exist an algorithm A that lists all Turing machines that halt on a blank tape input in some order of increasing lengths of the program. See if the original Turing machine is amongst the Turing machines generated by A. Since the length of the original program is fixed, the comparison will stop when this length is exceeded. Thus, we have a solution to the blank tape halting problem.

16. If the specific instances of the problem are $p_1, p_2, ..., p_n$, we construct a Turing machine that behaves as follows:

$$\begin{aligned} &\text{if problem } = p_1 \text{ then return false} \\ &\text{if problem } = p_2 \text{ then return true} \\ &\qquad\vdots \\ &\text{if problem } = p_n \text{ then return true} \end{aligned}$$

Whatever the truth values of the various instances are, there is always some Turing machine that gives the right answer. Remember that it is not necessary to know what the Turing machine actually is, only to guarantee that it exists.

Section 12.2

3. Suppose we had an algorithm to decide whether or not $L(M_1) \subseteq L(M_2)$. We could then construct a machine M_2 such that $L(M_2) = \varnothing$ and apply the algorithm. Then $L(M_1) \subseteq L(M_2)$ if and only if $L(M_1) = \varnothing$. But this contradicts Theorem 12.3, since we can construct M_1 from any given grammar G.

6. If we take $L(G_2) = \Sigma^*$, the problem becomes the question of Theorem 12.3 and is therefore undecidable.

8. Since there are some grammars for which $L(G) = L(G)^*$ and some for which this is not so, the undecidability follows from Rice's theorem. To do this from first principles is a little harder. Take the halting problem (M, w) and modify it (along the lines of Theorem 12.4), so that if (M, w) halts, $\widehat{M}$ will accept $\{a\}^*$ and if (M, w) does not halt, $\widehat{M}$ accepts $\varnothing$. From $\widehat{M}$ get the grammar $\widehat{G}$ by the construction leading to Theorem 11.7. If $L\left(\widehat{M}\right) = \{a\}^*$, then $L\left(\widehat{G}\right) = L\left(\widehat{G}\right)^* = \{a\}^*$. But if $L\left(\widehat{M}\right) = \varnothing$, then $L\left(\widehat{G}\right) = \varnothing$ and $L\left(\widehat{G}\right)^* = \{\lambda\}$. Therefore, if this problem were decidable, we could get a solution of the halting problem.

Section 12.3

1. A PC-solution is $w_3w_4w_1 = v_3v_4v_1$. There is no MPC-solution because one string would have a prefix 001, the other 01.

3. For a one-letter alphabet, there is a PC-solution if and only if there is some subset J of $\{1, 2, ..., n\}$ such that

$$\sum_{j \in J} |w_j| = \sum_{j \in J} |v_j| .$$

Since there are only a finite number of subsets, they can all be checked and therefore the problem is decidable.

5. (a) The problem is undecidable. If it were decidable, we would have an algorithm for deciding the original MPC-problem. Given $w_1, w_2 ...,$ w_n, we form $w_1^R, w_2^R ..., w_n^R$ and use the assumed algorithm. Since $w_1 w_i ... w_k = \left(w_k^R ... w_i^R w_1^R\right)^R$, the original MPC-problem has a solution if and only if the new MPC-problem has a solution.

Chapter 13

Section 13.1

2. Using the function *subtr* in Example 13.3, we get the solution

$$greater\,(x, y) = subtr\,(1, subtr\,(1, subtr\,(x, y))) .$$

7.

$$g\,(x, y) = mult\,(x, g\,(x, y - 1)) ,$$
$$g\,(x, 0) = 1.$$

9. (a)

$$\begin{aligned} A\,(1, y) &= A\,(0, A\,(1, y - 1)) \\ &= A\,(1, y - 1) + 1 \\ &= A\,(1, y - 2) + 2 \\ &\;\;\vdots \\ &= A\,(1, 0) + y \\ &= y + 2. \end{aligned}$$

(b) With the results of part (a) we can use induction to prove the next identity. Assume that for $y = 1, 2, ..., n-1$, we have $A(2, y) = 2y + 3$. Then

$$\begin{aligned} A(2, n) &= A(1, A(2, n-1)) \\ &= A(1, 2n+1) \\ &= 2n + 3, \text{ from part (a).} \end{aligned}$$

Since

$$\begin{aligned} A(2, 0) &= A(1, 1) \\ &= 3, \end{aligned}$$

we have a basis and the equation is true for all y.

15. If $2^x + y - 3 = 0$, then $y = 3 - 2^x$. The only values of x that give a positive y are 0 and 1, so the domain of μ is $\{0, 1\}$, giving a minimum value of $y = 1$. Therefore

$$\mu y\left(2^x + y - 3\right) = 1.$$

Section 13.2

1. (b) Use $C_T = \{a, b, c\}$, $C_N = \{x\}$ and $A = \{x\}$. The non-terminal x is used as a boundary between the left and right side of the target string and the two w's are built simultaneously by

$$V_1 x V_2 \to V_1 a x V_2 a \,|V_1 b x V_2 b|\, V_1 c x V_2 c.$$

At the end, the x is removed by

$$V_1 x V_2 \to V_1 V_2.$$

3. At every step, the only possible identification of V_1 is with the entire derived string. This results in a doubling of the string and

$$L = \left\{a^{2^n} : n \geq 1\right\}.$$

5. A solution is

$$\begin{aligned} V_1 * V_2 = V_3 &\to V_1 1 * V_2 = V_3 V_2 \\ V_1 * V_2 = V_3 &\to V_1 * V_2 1 = V_3 V_1. \end{aligned}$$

For example

$$1 * 1 = 1 \Rightarrow 11 * 1 = 11 \Rightarrow 11 * 11 = 1111,$$

and so on.

Section 13.3

1.

$$
\begin{aligned}
P_1 &: S \rightarrow S_1S_2 \\
P_2 &: S_1 \rightarrow aS_1, S_2 \rightarrow aS_2 \\
P_3 &: S_1 \rightarrow bS_1, S_2 \rightarrow bS_2 \\
P_4 &: S_1 \rightarrow \lambda, S_2 \rightarrow \lambda.
\end{aligned}
$$

5. The solution here is reminiscent of the use of messengers with context-sensitive grammars.

$$
\begin{aligned}
ab &\rightarrow x \\
xb &\rightarrow bx \\
xc &\rightarrow \lambda.
\end{aligned}
$$

8. Although this is not so easy to see, this is one way to solve Exercise 7. Take any string, say a^{255}. This can be derived from a^{127} by applying $a \rightarrow aaa$ once and $a \rightarrow aa$ 126 times. Then a^{127} can be derived from a^{63} in a similar way, and so on. Thus every string in $L(aa^*)$ can be derived.